SI-möte #6, Programkonstruktion och datastrukturer
Lösningsförslag

Elias Castegren & Kristiina Ausmees
elca7381@student.uu.se ‖ krau6498@student.uu.se

Övningar

1.

i) $T(n) = 2T(\frac{n}{2}) + n$

$a = 2, \qquad b = 2, \qquad f(n) = n$

$n^{log_2 2} = n, \qquad f(n) = \Theta(n) \implies$ **Case 2**

$T(n) = \Theta(n \cdot \log n)$

ii) $T(n) = 4T(\frac{n}{2}) + n^2 \cdot \lg n$

$a = 4, \qquad b = 2, \qquad f(n) = n^2 \cdot \lg n$

$n^{log_2 4} = n^2, \qquad \frac{n^2 \cdot \lg n}{n^2} = \lg n \neq \Omega(n^\epsilon) \implies$ **Master theorem ej applicerbart**

iii) $T(n) = 10T(\frac{n}{3}) + 2n^2$

$a = 10, \qquad b = 3, \qquad f(n) = 2n^2$

$n^{log_3 10} > n^2, \qquad \frac{n^{log_3 10}}{f(n)} = \Omega(n^\epsilon) \implies$ **Case 1**

$T(n) = \Theta(n^{log_3 10})$

iv) $T(n) = T(\frac{n}{2}) + n$

$a = 1, \qquad b = 2, \qquad f(n) = n$

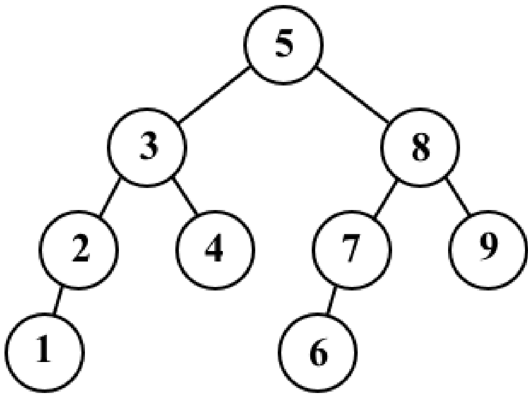
$n^{log_2 1} = n^0 = 1, \qquad \frac{f(n)}{1} = \Omega(n^\epsilon) \implies$ **Case 3**

$T(n) = \Theta(n)$

v) $T(n) = 2^n \cdot T(\frac{n}{2}) + n^4$

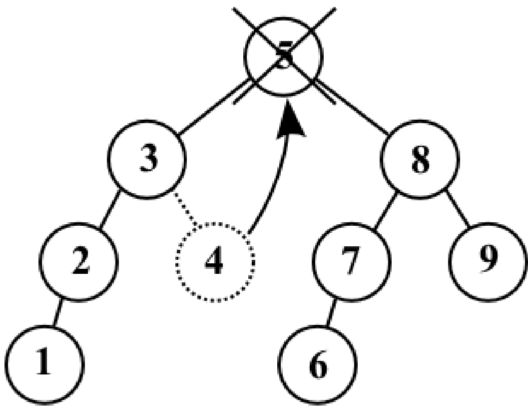
$a = 2^n$ inte en konstant $\implies$ **Master theorem ej applicerbart**

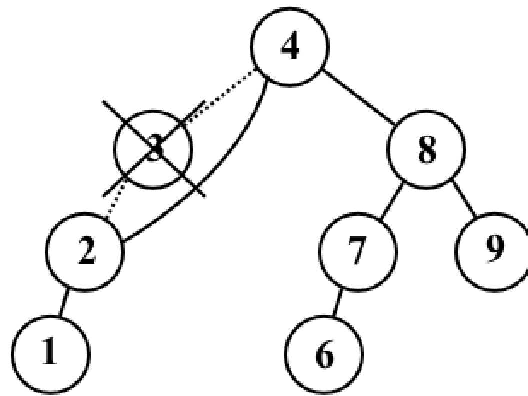
2.

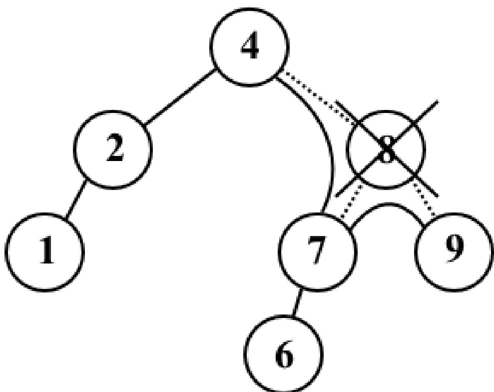


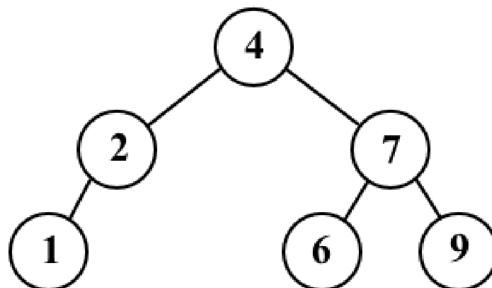
3.

När en nod tas bort i ett binärt sökträd måste den ersättas med den noden som har högst nyckel i det vänsta delträdet, eller den som har lägst nyckel i det högra delträdet. Nedan används den förstnämnda metoden:

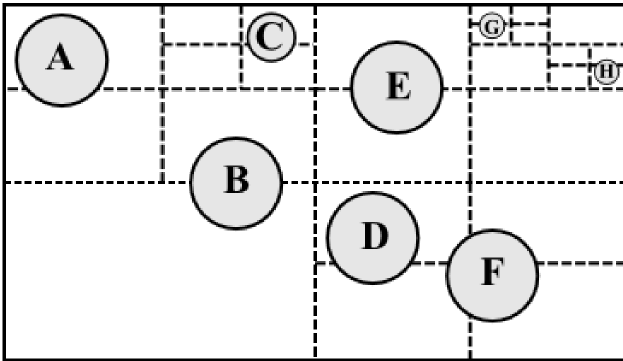




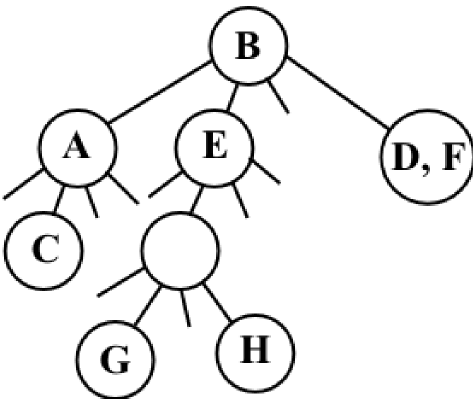




4.



När man ska rita quadträdet som representerar ovanstående figur är det bra att bestämma sig för en ordning på noderna. Nedan väljer vi att ordna en nods barn så att de från vänster till höger kopplas till förälderns övre vänstra, övre högra, nedre vänstra respektive nedre högra kvadrant:



Rotnoden innehåller som sagt bara cirkeln B och har ingen tredje undernod eftersom det inte finns några cirklar i den nedre vänstra kvadranten. Rotnodens fjärde barn har två cirklar i sig, eftersom både D och F skär områdets mittlinjer. En av noderna i trädet har inga cirklar i sig men måste ändå vara där eftersom nodens barn innehåller cirklar (G och H).

5.

```
datatype 'a tree = Void | Node of 'a * 'a tree * 'a tree;

(* treeHeight(t)
   TYPE: 'a tree -> int
   PRE: ()
   POST: höjden av t
   EXAMPLES: treeHeight(Node(1, Node(2, Void, Void) Void)) = 2
*)
(*VARIANT: antalet noder i t*)
fun treeHeight(Void) = 0
  | treeHeight(Node(_,L,R)) = 1 + Int.max(treeHeight(L), treeHeight(R));
```

Om man tänker lite på hur funktionen fungerar så inser man att varje nod i trädet kommer besökas exakt en gång, vilket tyder på att tidskomplexiteten borde vara $\Theta(n)$, där n är antalet noder i `t`. Om man antar att trädet är balanserat kan man ställa upp följande rekursion:

$$T(n) = \begin{cases} \Theta(1) & \text{om } n = 0 \\ 2T(\frac{n}{2}) + \Theta(1) & \text{om } n > 0 \end{cases}$$

där $\Theta(1)$ är tiden det tar att addera och hitta det största av två heltal. Applicerar man sedan Master Theorem får man att $n^{\log_2 2} = n^1 = n$, och eftersom det finns $\epsilon > 0$ så att $\frac{n}{k} = \Omega(n^\epsilon)$ för konstanta k (fall 1 av Master Theorem) så är tidskomplexiteten $\Theta(n)$ som väntat.

Om man antar att alla noder bara har en enda nod till höger eller vänster, alltså att det bara finns en enda väg genom trädet får man istället följande rekursion:

$$T(n) = \begin{cases} \Theta(1) & \text{om } n = 0 \\ T(n-1) + \Theta(1) & \text{om } n > 0 \end{cases}$$

där $\Theta(1)$ är tiden för addition och `Int.max` och för att på konstant tid undersöka det underträd som inte har några noder. Här är inte Master Theorem längre applicerbart, men man kan lätt visa att tidskomplexiteten fortfarande är $\Theta(n)$.

6.

pre-order: $[5, 3, 2, 1, 4, 8, 7, 6, 9]$

in-order: $[1, 2, 3, 4, 5, 6, 7, 8, 9]$

post-order: $[1, 2, 4, 3, 6, 7, 9, 8, 5]$

```
(* inOrder(t)
   TYPE: 'a tree -> 'a list
   PRE: ()
   POST: en lista med noderna i t i den ordning de besöks med en
         inorder-traversering
   EXAMPLES: inOrder(t) = [1,2,3,4,5,6,7,8,9]
             (där t är AVL-trädet från uppgift 2)

(*VARIANT: antalet noder i t*)
fun inOrder(Void) = []
  | inOrder(Node(e, L, R)) = inOrder(L) @ (e::inOrder(R));
```

Ifall det blir en pre-, in- eller postorder-traversering beror på var man sätter in elementet `e` i varje steg. För pre- och postorder-traverseringar skulle funktionskroppen bli
`e::inOrder(L) @ inOrder(R)`
respektive
`inOrder(L) @ inOrder(R) @ [e]`

I varje steg utförs en (eller två) listsammanslagning (append) med linjär tidskomplexitet. Om man antar att trädet är balanserat kan man ställa upp följande rekursion:

$$T(n) = \begin{cases} \Theta(1) & \text{om } n = 0 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{om } n > 0 \end{cases}$$

Applicerar man Master Theorem får man att $n^{\log_2 2} = n = \Theta(n)$ (fall 2 av Master Theorem), alltså att tidskomplexiteten är $T(n) = \Theta(n \log n)$. Eftersom varje nod besöks exakt en gång skulle man vilja att komplexiteten var linjär som i uppgift 4. Problemet ligger i listsammanslagningen som ger den irriterande $\Theta(n)$-termen i rekursionen. Ett sätt att bli av med den för en inorder-traversering är att besöka alla noder i omvänd ordning och i varje steg lägga in noden först i en ackumulatorlista:

```
fun inOrder'(Void, ack) = ack
  | inOrder'(Node(e, L, R), ack) = inOrder(L, e::inOrder(R, ack));

fun inOrder(t) = inOrder'(t, []);
```

Den resulterande rekursionen blir då istället

$$T(n) = \begin{cases} \Theta(1) & \text{om } n = 0 \\ 2T(\frac{n}{2}) + \Theta(1) & \text{om } n > 0 \end{cases}$$

Vilket ger $T(n) = \Theta(n)$ enligt samma resonemang som i uppgift 4.

7.

```
fun foo [x] = [x]
  | foo l = let
                val (a,b) = split(l)
            in
                foo(a) @ foo(b) @ foo(a) @ foo(b)
            end;
```

Om längden på listan `l` är n så kan vi beskriva längden på listan med följande rekursion:

$$L(n) = \begin{cases} 1 & \text{om } n = 1 \\ 4L(\frac{n}{2}) & \text{om } n > 1 \end{cases}$$

Den slutliga listan är ju resultatet av fyra rekursiva anrop ihopsatta till en lista. Antagandet att $n = 2^k$ gör att vi vet att `split` alltid kommer ge två listor med exakt halva längden av `l`, så problemstorleken för varje rekursivt anrop är alltid $\frac{n}{2}$.

Rekursionen känner vi igen som en på formen $L(n) = aT(\frac{n}{b}) + f(n)$, alltså kan vi försöka använda Master Theorem:

$$a = 4, \qquad b = 2, \qquad f(n) = 0$$

$$n^{log_2 4} = n^2, \qquad n^2 \text{ dominerar } 0 \implies \textbf{Case 1}$$

$$L(n) = \Theta(n^2)$$

Just för $n = 2^k$ blir längden exakt n^2, eftersom vi får exakta delningar med `split`, $f(n) = 0$ och basfallet är 1. För godtyckliga n kan vi i alla fall säga att den resulterande längden är i samma storleksordning som n^2, även om en exakt formel blir mer komplicerad.